CS61C Spring 2016 Discussion 3 – MIPS II/CALL

1 Translating between C and MIPS

Translate between the C and MIPS code. You may want to use the MIPS Green Sheet as a reference. We show you how the different variables map to registers – you don't have to worry about the stack or any memory-related issues.

C	MIPS

`// Nth_Fibonacci(n):`
`// $s0 -> n, $s1 -> fib`
`// $t0 -> i, $t1 -> j`
`// Assume fib, i, j are these values`
`int fib = 1, i = 1, j = 1;`
`if (n==0)  return 0;`
`else if (n==1) return 1;`
`n -= 2;`
`while (n != 0) {`
`  fib = i + j;`
`  j = i;`
`  i = fib;`
`  n--;`
`}`
`return fib;`
	`  ...`
	`  beq $s0, $0, Ret0`
	`  addiu $t2, $0, 1`
	`  beq $s0, $t2, Ret1`
	`  addiu $s0, $s0, -2`
	`Loop: beq $s0, $0, RetF`
	`  addu $s1, $t0, $t1`
	`  addiu $t0, $t1, 0`
	`  addiu $t1, $s1, 0`
	`  addiu $s0, $s0, -1`
	`  j  Loop`
	`Ret0: addiu $v0, $0, 0`
	`  j  Done`
	`Ret1: addiu $v0, $0, 1`
	`  j  Done`
	`RetF: addu $v0, $0, $s1`
	`Done: ...`

2 MIPS Addressing

- We have several **addressing modes** to access memory (immediate not listed):
  (a) **Base displacement addressing:** Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
  (b) **PC-relative addressing:** Uses the PC (actually the current PC plus four) and adds the I-value of the instruction (multiplied by 4) to create an address (used by I-format branching instructions like beq, bne)
  (c) **Pseudodirect addressing:** Uses the upper four bits of the PC and concatenates a 26-bit value from the instruction (with implicit 00 lowest bits) to make a 32-bit address (used by J-format instructions)
  (d) **Register Addressing:** Uses the value in a register as a memory address (jr)

1. You need to jump to an instruction that $2^{28} + 4$ bytes higher than the current PC. How do you do it? Assume you know the exact destination address at compile time. (Hint: you need multiple instructions)

   The jump instruction can only reach addresses that share the same upper 4 bits as the PC. A jump $2^{28}+4$ bytes away would require changing the fourth highest bit, so a jump instruction is not sufficient. We must manually load our 32 bit address into a register and use jr.

   ```
   lui $at {upper 16 bits of Foo}
   ori $at $at {lower 16 bits of Foo}
   jr $at
   ```

2. You now need to branch to an instruction $2^{17} + 4$ bytes higher than the current PC, when $t0 equals 0. Assume that we're not jumping to a new 2^{28} byte block. Write MIPS to do this.

   The largest address a branch instruction can reach is PC + 4 + SignExtImm. The immediate field is 16 bits and signed, so the largest value is $2^{15} - 1$ words, or $2^{17} - 4$ Bytes. Thus, we cannot use a branch instruction to reach our goal, but by the problem's assumption, we can use a jump. Assuming we're jumping to label Foo

```
bne $t0 $0 DontJump
j Foo
DontJump: ...
```

3. Given the following MIPS code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your green sheet!):

```
0x002cff00: loop: addu $t0, $t0, $t0      | 0 | 8 | 8 | 8 | 0 | 0x21 |
0x002cff04:       jal  foo                | 3 |      0xc0001         |
0x002cff08:       bne  $t0, $zero, loop   | 5 | 8 | 0 | -3 = 0xfffd  |
...
0x00300004: foo:  jr $ra                  $ra=__0x002cff08___
```

3 MIPS Calling Conventions

1. How should `$sp` be used? When do we add or subtract from `$sp`?

   `$sp` points to a location on the stack to load or store into. Subtract from `$sp` before storing, and add to `$sp` after restoring.

2. Which registers need to be saved or restored before using `jr` to return from a function?

   All `$s*` registers that were modified during the function must be restored to their value at the start of the function

3. Which registers need to be saved before using `jal`?

   `$ra`, and all `$t*`, `$a*`, and `$v*` registers if their values are needed later after the function call.

4. How do we pass arguments into functions?

   `$a0`, `$a1`, `$a2`, `$a3` are the four argument registers

5. What do we do if there are more than four arguments to a function?

   Use the stack to store additional arguments

6. How are values returned by functions?

   `$v0` and `$v1` are the return value registers.

4 Writing MIPS Functions

Here is a general template for writing functions in MIPS:

```
FunctionFoo:  # PROLOGUE
# begin by reserving space on the stack
addiu $sp, $sp, -FrameSize

# now, store needed registers
sw $ra, 0($sp)
sw $s0, 4($sp)
...
# BODY
...
# EPILOGUE
# restore registers
lw $s0 4($sp)
lw $ra 0($sp)

# release stack spaces
addiu $sp, $sp, FrameSize

# return to normal execution
jr $ra
```

Translate the following C code for a recursive function into a callable MIPS function.

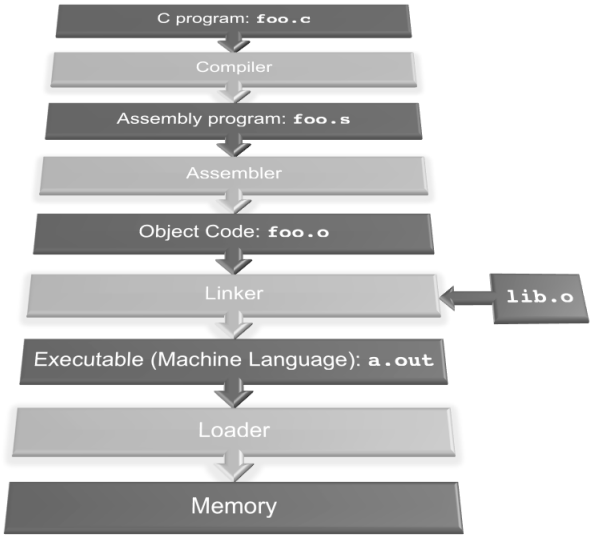
```
// Finds the sum of numbers 0 to N
int sum_numbers(int N) {
    int sum = 0

    if (N==0) {
        return 0;
    } else {
        return N + sum_numbers(N - 1);
    }
}
```

```
RecursiveSum:
addiu $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
li $v0, 0
beq $a0, $0, Ret
addiu $a0, $a0, -1
jal RecursiveSum
lw $a0, 0($sp)
addu $v0, $v0, $a0
Ret:
lw $ra, 4($sp)
addiu $sp, $sp, 8
jr $ra
```

5 Compile, Assemble, Link, Load, and Go!

5.1 Overview



5.2 Exercises

1. What is the Stored Program concept and what does it enable us to do?
   It is the idea that instructions are just the same as data, and we can treat them as such. This enables us to write programs that can manipulate other programs!

2. How many passes through the code does the Assembler have to make? Why?
   Two, one to find all the label adresses and another to convert all instructions while resolving any forward references using the collected label addresses.

3. What are the different parts of the object files output by the Assembler?
   Header: Size and position of other parts
   Text: The machine code
   Data: Binary representation of any data in the source file
   Relocation Table: Identifies lines of code that need to be "handled" by Linker
   Symbol Table: List of the files labels and data that can be referenced
   Debugging Information: Additional information for debuggers

4. Which step in CALL resolves relative addressing? Absolute addressing? Assembler, Linker.

5. What step in CALL may make use of the `$at` register? Assemble

6. What does RISC stand for? How is this related to pseudoinstructions?
   Reduced Instruction Set Computing. Minimal set of instructions leads to many lines of code. Pseudoinstructions are more complex instructions intended to make assembly programming easier for the coder. These are converted to TAL by the assembler.